Faster FPTASes for counting and random generation of Knapsack solutions

Rizzi, Romeo

2019-08

Rizzi , R & Tomescu , A I 2019 , ' Faster FPTASes for counting and random generation of Knapsack solutions ' , Information and Computation , vol. 267 , pp. 135-144 . https://doi.org/10.1016/j.ic.2019.04.001

http://hdl.handle.net/10138/302575
https://doi.org/10.1016/j.ic.2019.04.001

Information and Computation 267 (2019) 135–144

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Faster FPTASes for counting and random generation of Knapsack solutions ☆

Romeo Rizzi [a], Alexandru I. Tomescu [b,*]

[a] Department of Computer Science, University of Verona, Italy
[b] Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland

ARTICLE INFO

Article history:
Received 30 March 2015
Received in revised form 14 February 2017
Available online 10 April 2019

Keywords:
0/1 Knapsack problem
Approximation algorithm
Counting problem
Sampling
Dynamic programming
Directed acyclic graph

ABSTRACT

In the #P-complete problem of counting 0/1 Knapsack solutions, the input consists of a sequence of n nonnegative integer weights $w_1, \ldots, w_n$ and an integer C, and we have to find the number of subsequences (subsets of indices) with total weight at most C. We give faster and simpler fully polynomial-time approximation schemes (FPTASes) for this problem, and for its random generation counterpart. Our method is based on dynamic programming and discretization of large numbers through floating-point arithmetic. We improve both deterministic counting FPTASes from Gopalan et al. (2011) [9], Štefankovič et al. (2012) [6] and the randomized counting and random generation algorithms in Dyer (2003) [5].

Our method is general, and it can be directly applied on top of combinatorial decompositions (such as dynamic programming solutions) of various problems. For example, we also improve the complexity of the problem of counting 0/1 Knapsack solutions in an arc-weighted DAG.

1. Introduction

The 0/1 Knapsack counting problem is defined as follows. The input consists of a sequence of n nonnegative integer weights $w_1, \ldots, w_n$ and an integer capacity C. A 0/1 Knapsack *solution* is a subset of the indices $\{1, \ldots, n\}$ whose associated weights add up to at most C. The counting problem asks for the number of solutions to a 0/1 Knapsack instance. Because this problem is #P-hard, a long line of research has focused on faster algorithms that find only approximate answers to it. More specifically, in algorithms that output a number that has relative error of $1 \pm \varepsilon$ with respect to the exact one, and run in time polynomial in the input size and in $1/\varepsilon$. Such an algorithm is called a *fully polynomial-time approximation scheme (FPTAS)*, or, if it is randomized, a *fully polynomial-time randomized approximation scheme (FPRAS)*.

In [2], the 0/1 Knapsack problem was extended to a directed acyclic graph (DAG) with nonnegative arc weights, in connection to various applications in biological sequence analysis (see the references in [2]). Given two vertices s and t, we have to count the number of s,t-paths of total weight at most C. This is a generalization because given an instance $w_1, \ldots, w_n$ and C it suffices to construct the DAG having vertex set $\{v_0, \ldots, v_n\}$, $s = v_0$, $t = v_n$, and for each $i \in \{1, \ldots, n\}$, there are two parallel arcs from v_{i-1} to v_i, with weights 0 and w_i, respectively.

☆ This paper is an updated and extended version of the ESA 2014 paper [1]. Partial support came from the Academy of Finland under grant 274977.
* Corresponding author.
E-mail addresses: romeo.rizzi@univr.it (R. Rizzi), alexandru.tomescu@helsinki.fi (A.I. Tomescu).

https://doi.org/10.1016/j.ic.2019.04.001

We are also interested in the *random generation* (or *sampling*) version of the 0/1 Knapsack problem. That is, given an input for 0/1 Knapsack, we need to generate one solution uniformly at random among all solutions. It follows from general results such as [3] that for a large class of problems (including 0/1 Knapsack) counting and random generation are inter-reducible. As such, also random generation of 0/1 Knapsack solutions is hard and we are thus interested in faster algorithms that generate a solution with near-uniform probability. In fact, the first algorithms for approximately counting 0/1 Knapsack solutions were obtained by near-uniform sampling, as we will review below.

The difficulty of counting 0/1 Knapsack solutions lies dually either in the size of the capacity or in the number of solutions. Previous algorithms such as [4] and [5] adopted the former perspective by discretizing capacities. Recently, [6] switched the perspective to discretizing the number of solutions, and obtained the fastest FPTAS to date. Adopting the same perspective, this paper gives three main contributions:

- We show that a natural approach to discretizing the large numbers of solutions is to use floating-point numbers whose mantissa is just as long to guarantee the final $1+\varepsilon$ approximation ratio (Section 2). We correlate the length of the mantissa with the number of steps performed by the counting algorithm, and develop a general floating-point approximation layer that can be plugged-in to other problems.
- Using such floating-point numbers, and together with a different organization of the counting computation, we are able to improve the FPTAS of [6] by a $O(\log n)$ factor (Section 3). This is also simpler, and thanks to the floating-point approximation layer, has a simpler approximation analysis. We also obtain a faster FPTAS for counting 0/1 Knapsack solutions on a DAG (Section 5).
- Both of the above insights allow us to obtain the first Las Vegas algorithm for near-uniform sampling of 0/1 Knapsack solutions (Section 4). This is also faster than the previous Monte Carlo near-uniform sampling algorithms.

1.1. Overview of previous results

Dyer [7] was the first to propose an algorithm based on near-uniform sampling of feasible solutions by a random walk, which works in subexponential time. Later, Morris and Sinclair [8] gave a Markov chain algorithm and showed that it is rapidly mixing, obtaining thus an FPRAS.

In [5], Dyer gave the first solution based on dynamic programming (and not on Markov chains), but still requiring sampling. This works by first constructing a Monte Carlo algorithm that nearly uniformly generates a solution with probability at least $1-e^{-1}$, using $O(n^2)$ arithmetic operations, once a supporting table is computed with $O(n^3)$ arithmetic operations. In this table, the capacities are discretized to n^2 values in total. By generating enough samples, and rejecting those that are not solutions to the 0/1 Knapsack problem, Dyer obtains a counting FPRAS using $O(n^3+n^2\varepsilon^{-2})$ arithmetic operations.

However, it is crucial to point out that these arithmetic operations are on $O(n)$-bit numbers since there can be $O(2^n)$ solutions. Thus, the Monte Carlo algorithm generates ν samples in time $O(n^4+\nu n^3)$, and the counting FPRAS runs in time $O(n^4+n^3\varepsilon^{-2})$.

Gopalan et al. [4] and Štefankovič et al. [6] (see also the combined paper [9]) were the first to give deterministic counting FPTASes that do not use sampling. Both are also based on dynamic programming.

Gopalan et al. modeled the problem as a read-once branching program, having a state (i,c), for each layer $i=1,\ldots,n$, and each partial sum $c=\sum_{j=1}^{i} a_j w_j \leq C$ (for some binary vector $a\in\{0,1\}^i$). Keeping the technicalities to a minimum, we can say that each state is associated with the number of paths in the program leading to an accepting final state (n,c) for $c\leq C$.[1] The approximation is achieved by discretizing the capacities, which is the same strategy of [5]. More precisely, the states in layer i are partitioned into intervals according to their second component $c\leq C$ so that all states in the same interval have approximately the same number of accepting paths. Because the number of exact states at any layer is $O(C)$, this inherently adds a dependence on C when computing numbers associated to each state. Since the numbers are monotonic as c increases, this dependence can be limited to a factor $O(\log C)$ by using a binary tree. Gopalan et al. also argue that their approximated branching program can be used to write a Monte Carlo near-uniform sampling algorithm.

Gopalan et al. state that the final time complexity of their counting FPTAS is $O(n^2\varepsilon^{-1}\log(n/\varepsilon)\log C)$, also assuming unit-cost additions of arbitrarily large integers. However, the integers appearing during the computation have $O(n)$ bits. As such, in order to directly compare their complexity bound with ours, we need to add another $O(n)$ factor to their complexity bound. We should also note that both techniques from [5] and [4] are general, and can be applied to other problems, such as multidimensional Knapsack, general integer Knapsack and contingency tables.

Štefankovič et al. [6] were the first to discretize the *number* of solutions, and not the capacities like in [5] and [4]. As such, they manage to remove the dependence on C, obtaining an FPTAS running in time $O(n^3\varepsilon^{-1}\log(n/\varepsilon))$. However, unlike [5] and [4], this method does not appear as general, as it has not been shown to be applicable to the generalized versions of Knapsack mentioned above.

This FPTAS is based on the decomposition $\tau(i,a)$:= the smallest capacity c such that there exist at least a solutions to the 0/1 Knapsack problem with weights $w_1,\ldots,w_i$ and capacity c. The second parameter of τ is then approximated according

[1] To be precise, it is associated with the probability of that state leading to an accepting final state.

to a geometric progression of ratio $Q = 1 + \varepsilon/(n+1)$. The approximated table is computed by dynamic programming using the recurrence:

$$T[i, j] = \min_{\alpha \in [0,1]} \max \begin{cases} T[i-1, \lfloor j + \log_Q \alpha \rfloor], \\ T[i-1, \lfloor j + \log_Q (1-\alpha) \rfloor] + w_i. \end{cases} \tag{1}$$

Thanks to the geometric discretization, the second parameter of T takes $O(n^2\varepsilon^{-1})$ values. Finding the minimum over $\alpha \in [0, 1]$ is reducible to two binary searches in row $i-1$ of T, due to its monotonicity. The computation of $\lfloor \log_Q \alpha \rfloor$ and $\lfloor \log_Q (1-\alpha) \rfloor$ is then shown to be doable in time $O(\log(n/\varepsilon))$.

In [2], the technique of Štefankovič et al. [6] was extended to counting 0/1 Knapsack solutions on a DAG, leading to an FPTAS running in time $O(mn^3 \log(n)\varepsilon^{-1})$.

1.2. Our results and approach

In this section we give a brief overview of our results and approach. Our main result is the following FPTAS.

Theorem 1. *Let $w_1, \ldots, w_n$ and C be an input to the 0/1 Knapsack counting problem, and let Z be the number of solutions. For any $0 < \varepsilon \leq 1$ we can deterministically compute a floating-point number Z' with a $\lceil \log n \rceil$-bit exponent and a $\lceil \log n + \log(1/\varepsilon) + 1 \rceil$-bit mantissa satisfying $(1-\varepsilon)Z \leq Z' \leq Z$, in time*

$$O(n^3 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil),$$

assuming unit cost additions and comparisons on numbers with $O(\log C)$ bits.[2]

This result is based on the classic decomposition over the family of subproblems $s(i, c)$:= the number of 0/1 Knapsack solutions that use a subset of the items $w_1, \ldots, w_i$, and their weights sum up to at most $c \leq C$. Notice that s and τ are dual, in the sense

$$\tau(i, a) = \min\{c \,:\, s(i, c) \geq a\} \text{ and } s(i, c) = \max\{a \,:\, \tau(i, a) \leq c\}. \tag{2}$$

Table s can be computed by a dynamic programming algorithm obtained from the recurrence

$$s(i, c) = s(i-1, c) + s(i-1, c-w_i). \tag{3}$$

We also approximate the number of solutions, which are now the values of $s(\cdot, \cdot)$. However, we approximate them using binary floating-point numbers. The difference with respect to a standard computer implementation is that we need as many bits for the exponent as to represent them exactly, and as many bits for the mantissa as to guarantee the required approximation. For counting 0/1 Knapsack solutions, we need $\lceil \log n \rceil$ bits for the exponent and $1 + \lceil \log(n/\varepsilon) \rceil$ bits for the mantissa. The main advantage of such an approximation scheme for the values in table s is that it avoids computing values such as $\lfloor \log_Q \alpha \rfloor$ (and the associated complexity analysis), and requires a much simpler approximation analysis.

Second, we are able to avoid a minimization as in (1), and thus avoid binary search. The idea is to store each row of s as a list. We prune entries with the same value in each list (leading to $O(n^2/\varepsilon)$ different entries in each list), and compute a list by two linear scans of the previous one.

Recurrence relation (3) can be extended to a DAG, and thus we can analogously obtain a counting FPTAS for this problem.

Theorem 2. *Let G be a DAG with n vertices, m arcs with nonnegative weights, and let s and t be two of its vertices. For any C and any $0 < \varepsilon \leq 1$, we can deterministically compute an $1-\varepsilon$ approximation of the number of s, t-paths of weight at most C in G, in time*

$$O\left(mn^2 \log\left(1 + \frac{m}{n}\right) \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil\right),$$

assuming unit cost additions and comparisons on numbers with $O(\log C)$ bits.

Another ingredient of the algorithm from Theorem 2 with respect to [2] is the organization of the computation in sequences of $O(n \log(1 + \frac{m}{n}))$ successive additions. As such, we need floating-point numbers with only $1 + \lceil \log(n \log(1 + \frac{m}{n})/\varepsilon) \rceil$ bits for the mantissa, and $\lceil \log n \rceil$ bits for the exponent.

Finally, having the table s explicitly, we can implement a Las Vegas near-uniform sampling algorithm of 0/1 Knapsack solution.[3] This works by tracing back probabilistically a random solution from $s(n, C)$. We obtain the following result.

[2] Our assumption that additions and comparisons of $O(\log C)$-bit numbers take unit time is also made in [7,9,6].

[3] Our notion of near-uniform sampling is the same as in e.g., [3, Section 6].

Theorem 3. *Let $w_1, \ldots, w_n$ and C be an input to the 0/1 Knapsack problem. For any $0 < \varepsilon \leq 1$, we can generate a solution with a probability different from the uniform one by a relative factor $(1-\varepsilon)^{\pm 1}$, in expected time*

$$O(n \log(n/\varepsilon)).$$

This assumes data structures occupying $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil W)$ bits and computable in $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil M(\lceil \log(1/\varepsilon)/\log n \rceil)$ time, where

- *W is the word size, i.e., the number of bits used to store a pointer; it suffices $W = \Omega(\log(nC))$ and $W = O(\lceil \log(n/\varepsilon) \rceil)$;*
- *$M(x)$ denotes the multiplicative slowdown of multiplying two x-bit numbers[4];*
- *we assume additions and comparisons of $O(\log C)$-bit numbers take constant time.*

Observe that this algorithm always returns a correct solution, as opposed to the Monte Carlo sampling algorithms from [5] and [4]. Our supporting data structures are computed slower than in [5]; however, the time needed for generating one solution is smaller by orders of magnitude.

At each step i, our algorithm throws a dice with two faces of sizes $s(i-1, c)$ and $s(i-1, c-w_i)$, where c is the capacity available for the remaining first i items. In order to guarantee that the sampling distribution differs from the uniform one by a factor $(1-\varepsilon)^{\pm 1}$, we need another $\lceil \log n \rceil$ bit for the mantissa of our approximated floating-point numbers, as this algorithm makes n choices. Since we represent the table s as a collection of lists, we need to keep, for every entry of a list, back-pointers to the corresponding two entries in the previous list. Each such a pointer occupies W bits, where W is the word size. Since our table has $O(n^3/\varepsilon)$ different entries in each list, we need $O(\lceil \log(n/\varepsilon) \rceil)$ bits for each pointer.

Notice that the possibility of doing random generation presented itself also in [6]. First, one needs to decrease Q to $Q = 1 + \varepsilon/(n(n+1))$ (as we do by increasing the length of the mantissa). Thanks to equations (2), one could employ the same probabilistic trace back, by using the approximated table T as black box, and decoding each necessary value of s from T. This can be done in time $O(\log(n/\varepsilon))$ by doing binary search in the corresponding row of T, which adds another factor $\log(n/\varepsilon)$ to the construction time. However, this can be avoided by similarly storing back-pointers from each entry of T to the corresponding two entries in the previous row of T, which are obtained when having found the minimum over $\alpha \in [0, 1]$. Otherwise put, the two faces of the approximated dice will have sizes $\lfloor j + \log_Q \alpha \rfloor$ and $\lfloor j + \log_Q (1-\alpha) \rfloor$, where α minimizes (1).

A further issue is rolling this dice in (expected) time proportional to the number of bits of the two faces of the approximated dice. In our case, since the two faces are floating-point numbers, we can easily solve this by generating a random floating-point number x as follows. We generate a sequence of bits until seeing the first bit equal to '1'. The expected number of bits until this happens is 2, thus this charges only a tiny $O(1)$ term to the expected value of the running time of rolling one dice. At this point, we know the exponent of x, and it is sufficient to continue generating only the remaining bits of the mantissa of x, and check whether x is smaller than the ratio between the approximations of $s(i-1, c-w_i)$ and $s(i, c)$. Moreover, our improvement and simplification obtained by Theorem 1 preserves itself in this random generation algorithm.

2. Approximation by floating-point numbers

In this paper, floating-point arithmetic with base 2 is sufficient, as it also has the advantage of being immediately implementable on a computer for small enough instances. Floating-point arithmetic, and the inherent accuracy analysis issues, have a long history in numerical computation. Another recent application of floating-point arithmetic to approximate counting problems was in [11] in connection with uniform random generation of decomposable structures by partial approximate counts. Moreover, observe that, conceptually, floating-point arithmetic can be seen as an effective combination of the geometric discretization of [6], through the exponent, and of the linear discretization of [7], through the mantissa.

Throughout this paper, we assume that the problem instances consist of n objects (0/1 Knapsack instances with n objects, DAGs with n vertices). Let $k \geq 1$ be such that the maximum numerical value of a particular counting problem is $2^{n^k} - 1$ (that is, it can be represented with n^k bits). Any number $x \in \{0, \ldots, 2^{n^k} - 1\}$ can be written as

$$x = x_1 2^{p-1} + x_2 2^{p-2} + \cdots + x_{p-1} 2^1 + x_p 2^0 = 2^p \left(x_1 2^{-1} + x_2 2^{-2} + \cdots + x_p 2^{-p} \right),$$

where $1 \leq p \leq n^k$, $x_1 = 1$, and $x_i \in \{0, 1\}$, for $i \in \{2, \ldots, p\}$. Under floating-point arithmetic terminology, p is called the *exponent* of x, and the binary string $x_1 x_2 \ldots x_p$ is called its *mantissa*.

We will approximate x as a floating-point number which has $k \log n$ bits dedicated to store its exponent p exactly, but only t bits dedicated to store the first t bits of its mantissa; that is, we approximate x by the number

$$\langle x \rangle_{k \log n, t} := 2^p \left(x_1 2^{-1} + x_2 2^{-2} + \cdots + x_t 2^{-t} \right).$$

[4] With the Schönhage-Strassen method [10], two x-bit numbers can be multiplied in time $x \log x \log\log x$, thus $M(x) = \log x \log\log x$.

We will often drop the subscript $k\log n, t$ when this will be clear from the context. We will always choose $t \geq k\log n$ since the contrary cannot help in reducing the memory consumption (and time), as we approximate only by shortening the mantissa while the exponent is always represented in full.

For every $0 \leq x < 2^{n^k}$, it holds that

$$(1-2^{1-t})x \leq \langle x \rangle_{k\log n,t} \leq x. \tag{4}$$

Let $\underline{x}$ and $\underline{y}$ be two floating-point numbers with $k\log n$ bits for the exponent and t bits for the mantissa. We denote the sum $\left\langle \underline{x}+\underline{y} \right\rangle$ by $\underline{x} \oplus \underline{y}$. We assume that we can compute $\underline{x} \oplus \underline{y}$ with a bit complexity of $O(k\log n + t) = O(t)$; if additions on $O(\log n)$-bit numbers take unit time, then we assume we can compute $\underline{x} \oplus \underline{y}$ with a word complexity of $O(t/\log n)$.

If $x, y \in \{0, \ldots, 2^{n^k}-1\}$ are such that $x+y \in \{0, \ldots, 2^{n^k}-1\}$, and $\underline{x}, \underline{y}$ are two floating-point numbers with $k\log n$ bits for the exponent and t bits for the mantissa such that

$$(1-2^{1-t})^i x \leq \underline{x} \leq x, \text{ and } (1-2^{1-t})^j y \leq \underline{y} \leq y,$$

for some integers $i, j \geq 0$, then by (4) the following inequality holds

$$(1-2^{1-t})^{1+\max(i,j)}(x+y) \leq \underline{x} \oplus \underline{y} \leq x+y, \tag{5}$$

For each particular problem, we will choose t as a function of n and of the error factor ε, $0 < \varepsilon \leq 1$. For the problem of counting 0/1 Knapsack solutions, $k=1$ and $t(n,\varepsilon) = 1 + \lceil \log(n/\varepsilon) \rceil$, while for its extension on a DAG, $k=1$ and $t(n,\varepsilon) = 1 + \lceil \log(n \log(1+\frac{m}{n})/\varepsilon) \rceil$. For the random generation of 0/1 Knapsack solutions, $k=1$ and $t(n,\varepsilon) = 1 + \lceil \log(n^2/\varepsilon) \rceil$.

3. Counting 0/1 Knapsack solutions

The classic pseudo-polynomial algorithm for counting 0/1 Knapsack solutions defines $s(i,c)$ as the number of Knapsack solutions that use a subset of the items $\{1, \ldots, i\}$, of weight at most $c \in \{0, \ldots, C\}$, and computes these values $s(i,c)$ by dynamic programming, using the recurrence

$$s(i,c) = s(i-1,c) + s(i-1, c-w_i), \tag{6}$$

where $s(0,c) = 1$ for any $c \geq 0$, and $s(i,c) = 0$, for any $i \in \{1, \ldots, n\}$ and $c < 0$. Indeed, we either use only a subset of items from $\{1, \ldots, i-1\}$ whose weights sum up to c, or use item i of weight w_i and a subset of items from $\{1, \ldots, i-1\}$ whose weights sum up to $c - w_i$. This DP algorithm executes nC additions on n-bit numbers and its complexity is $O(Cn^2)$. When $C \leq n$, this complexity becomes $O(n^3)$. This is a strongly polynomial bound and thus there is no need for an approximation algorithm. Hence, we will assume $n \leq C$ in what follows. We will assume, like in [6], that additions and comparisons on numbers with $O(\log C)$ bits have unit cost, which implies the same on $O(\log n)$-bit numbers.

We use relation (6) to count, but our numbers, for any $0 < \varepsilon \leq 1$, are approximate floating-point numbers with $\lceil \log n \rceil$ bits for the exponent, and $1 + \lceil \log(n/\varepsilon) \rceil$ bits for the mantissa (we can assume for simplicity that a solution using all n objects has cost greater than C, so that $s(i,c) < 2^n$ for all $i \in \{1, \ldots, n\}$, $c \in \{0, \ldots, C\}$). By the above assumption, we have that additions and comparisons of these floating-point numbers on $O(\log(n/\varepsilon))$ bits take time $O(\lceil \log(n/\varepsilon)/\log n \rceil) = O(\lceil \log(1/\varepsilon)/\log n \rceil)$.

For every $i \in \{0, \ldots, n\}$ we keep a list, $list(i)$, whose entries are pairs of the form $[c, \underline{r}]$, where c is a capacity in $\{0, \ldots, C\}$ and $\underline{r}$ is an approximate floating-point number of solutions. Having $list(i)$, for every $c \in \{0, \ldots, C\}$ we define $\underline{s}(i,c) := \max\{\underline{r} : [c', \underline{r}] \in list(i),\ c' \leq c\}$, where the maximum of an empty set is taken to be 0. We will refer to the set of first components of the pairs in $list(i)$ as the *capacities in* $list(i)$.

The first list, $list(0)$, consists of the single pair $[0, 1]$. After this initialization, while computing $list(i)$ from $list(i-1)$, we maintain the following two invariants:

(I_1) $list(i)$ is strictly increasing on both components;
(I_2) $(1-\varepsilon/n)^i s(i,c) \leq \underline{s}(i,c) \leq s(i,c)$, for every $c \in \{0, \ldots, C\}$.

Note that Property (I_1) implies that the length of $list(i)$ is at most the total number of floating-point numbers that can be represented with $\lceil \log n \rceil + \lceil \log(n/\varepsilon) \rceil + 1$ bits, that is $O(n^2/\varepsilon)$.

We obtain $list(i)$ by first building a list, which we denote $list'(i)$, that, for every capacity c in $list(i-1)$, contains the following two pairs:

$$[c, \underline{s}(i-1,c) \oplus \underline{s}(i-1, c-w_i)] \text{ and } [c+w_i, \underline{s}(i-1, c+w_i) \oplus \underline{s}(i-1,c)]. \tag{7}$$

We say that a pair $[c_1, \underline{r}_1]$ is *dominated* by a pair $[c_2, \underline{r}_2]$ if $c_2 \leq c_1$ and $\underline{r}_2 \geq \underline{r}_1$. From $list'(i)$ we remove all dominated pairs, and by sorting $list'(i)$ on any of the two components we can obtain $list(i)$ satisfying Property (I_1). In fact, Lemma 1 below shows that we can avoid sorting $list'(i)$, and we can compute it in linear time from $list(i-1)$.

We summarize the entire approximate counting procedure as Algorithm 1.

Algorithm 1: ApproximateCount$(w_1, \ldots, w_n, C)$. An FPTAS for counting 0/1 Knapsack solutions.

Notation: $\underline{s}(i,c) := \max\{r : [c', \underline{r}] \in list(i), c' \leq c\}$;
insert the pair $[0,1]$ into $list(0)$;
for $i = 1$ **to** n **do**
  construct the bimonotonic $list'(i)$ containing, for each $[c, \underline{r}]$ in $list(i-1)$, the two pairs:
    - $[c, \underline{r} \oplus \underline{s}(i-1, c-w_i)]$;
    - $[c+w_i, \underline{s}(i-1, c+w_i) \oplus \underline{r}]$;
  obtain $list(i)$ by scanning $list'(i)$ and removing dominated pairs;
return $\underline{s}(n, C)$.

Lemma 1. *We can compute $list(i)$ from $list(i-1)$ in time $O(n^2\varepsilon^{-1}\lceil\log(1/\varepsilon)/\log n\rceil)$.*

Proof. At a generic step $i \in \{1, \ldots, n\}$, we compute $list'(i)$ as follows. We construct two auxiliary lists of pairs, $back(i)$ and $forw(i)$. For every capacity c in $list(i-1)$, the list $back(i)$ will contain the pairs $[c, \underline{s}(i-1,c) \oplus \underline{s}(i-1, c-w_i)]$, and the list $forw(i)$ will contain the pairs $[c+w_i, \underline{s}(i-1, c+w_i) \oplus \underline{s}(i-1,c)]$. List $list'(i)$ is now obtained by merging in a unique sorted list the lists $back(i)$ and $forw(i)$.

In order to compute $forw(i)$, proceed as follows (the computation of $back(i)$ is entirely analogous). Keep two pointers $left$ and $right$ in $list(i-1)$. Pointer $left$ is initially set to the first pair in $list(i-1)$, say $[c, \underline{r}]$. Pointer $right$ is also set to the first pair in $list(i-1)$, but starts scanning $list(i-1)$ until reaching a pair $[c_1, \underline{r}_1]$, such that $c_1 + w_i \geq c$ and either $[c_1, \underline{r}_1]$ is the last pair in $list(i-1)$, or $[c_1, \underline{r}_1]$ is immediately followed by a pair $[c_2, \underline{r}_2]$ with the property $c + w_i < c_2$. Append the pair $[c+w_i, \underline{r}_1 \oplus \underline{r}]$ at the end of $forw(i)$, and advance pointer $left$ to the next pair in $list(i-1)$; repeat the above procedure, by advancing pointer $right$ to the corresponding pair, and inserting a new resulting pair in $forw(i)$. This is repeated until pointer $left$ reaches the end of $list(i-1)$.

Observe that list $forw(i)$ is bimonotonic (i.e. non-decreasing on both components), by the fact that Property (I_1) holds for $list(i-1)$. By analogy, this is true also for $back(i)$. Therefore, we can merge them on the first component, making sure that we also remove dominated pairs. That is, whenever during merging we compare a pair $[c_1, \underline{r}_1]$ from $forw(i)$ with a pair $[c_2, \underline{r}_2]$ from $back(i)$, with $c_2 \leq c_1$: if $\underline{r}_2 < \underline{r}_1$, then we copy $[c_2, \underline{r}_2]$ into $list(i)$ as usual; otherwise, if $\underline{r}_2 \geq \underline{r}_1$, we drop the pair $[c_2, \underline{r}_2]$. Thus Property (I_1) holds for $list(i)$.

Since we assume that additions and comparisons on $O(\log C)$-bit numbers take unit time, that floating-point additions and comparisons take $O(\lceil\log(1/\varepsilon)/\log n\rceil)$ time, and the length of $list(i-1)$ is $O(n^2/\varepsilon)$, the construction of $list(i)$ takes time $O(n^2\varepsilon^{-1}\lceil\log(1/\varepsilon)/\log n\rceil)$. □

Lemma 2. *Property (I_2) holds for $list(i)$, that is, for every $i \in \{0, \ldots, n\}$ and every $c \in \{0, \ldots, C\}$, $(1-\varepsilon/n)^i\, s(i,c) \leq \underline{s}(i,c) \leq s(i,c)$ holds.*

Proof. The claim is clear for $i = 0$. For an arbitrary capacity $c \in \{0, \ldots, C\}$, let $[c_1, \underline{r}_1]$ in $list(i)$ be such that $\underline{s}(i,c) = \underline{r}_1$. From the definition of $\underline{s}$, we get $\underline{s}(i,c) = \underline{s}(i,c_1)$; from the fact that the pairs in $list(i)$ are of the form (7), we have

$$\underline{s}(i,c) = \underline{s}(i,c_1) = \underline{s}(i-1,c_1) \oplus \underline{s}(i-1, c_1 - w_i). \tag{8}$$

We now prove two properties about the two terms of the above $\oplus$ sum.

Claim 1. $\underline{s}(i-1,c_1) = \underline{s}(i-1,c)$.

Proof of Claim 1. The capacities in $list(i-1)$ are a subset of the capacities in $list'(i)$. Also, we have pruned the pairs in $list'(i)$ by keeping the smallest capacity for every approximate number of solutions corresponding to that capacity. Thus, since $\underline{s}(i,c_1) = \underline{s}(i,c)$ holds, then also $\underline{s}(i-1,c_1) = \underline{s}(i-1,c)$ holds. □

Claim 2. $\underline{s}(i-1, c_1 - w_i) = \underline{s}(i-1, c-w_i)$.

Proof of Claim 2. It suffices to show that there is no capacity c_2 in $list(i-1)$ such that $c_1 - w_i < c_2 < c - w_i$. Indeed, for assuming the contrary, $c_2 + w_i$ would be a capacity in $list'(i)$, by (7). Since we have chosen c_1 as the largest capacity in $list(i)$ smaller than c, and $c_1 < c_2 + w_i < c$ holds, this implies that $c_2 + w_i$ was pruned when passing from $list'(i)$ to $list(i)$; thus, the two pairs of $list'(i)$ having c_1 and $c_2 + w_i$ as first components have equal second components. By (8) and the fact that Property (I_1) holds for $list(i-1)$, this entails that also the two pairs of $list(i-1)$ having $c_1 - w_i$ and c_2 as first components must have equal second components. This contradicts the fact that $list(i-1)$ satisfies Property (I_1). □

Plugging the equations from Claims 1 and 2 into (8) we obtain

$$\underline{s}(i,c) = \underline{s}(i-1,c) \oplus \underline{s}(i-1, c-w_i). \tag{9}$$

From (6), the fact that Property (I_2) holds for $list(i-1)$, and from (5), we get that $(1-\varepsilon/n)^i s(i,c) \leq \underline{s}(i,c) \leq s(i,c)$, which shows that Property (I_2) holds also for $list(i)$ and completes the proof. □

By standard techniques, for all natural numbers $n \geq 1$ and all $0 < \varepsilon \leq 1$, the following hold:

$$1-\varepsilon \leq \left(1-\frac{\varepsilon}{n}\right)^n, \text{ and } \left(1-\frac{\varepsilon}{n}\right)^{-n} \leq (1-\varepsilon)^{-1}. \tag{10}$$

From Lemma 2, the fact that Property (I_2) holds, and (10), we obtain Theorem 1.

4. Random generation of 0/1 Knapsack solutions

For the random generation problem, we increase the length of the mantissa of the floating-point numbers up to $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits.

Let $\underline{f}(i,c) := \underline{s}(i-1, c-w_i) \oslash \underline{s}(i,c)$ ('$\oslash$' denotes the floating-point division).[5] It is important to remark here that the number of solutions including object i is at most the number of solutions not including object i. Indeed, to every solution S containing object i we can associate a different solution not containing object i, namely $S \setminus \{i\}$. It follows that $\underline{f}(i,c) \leq \frac{1}{2}$ so that $\underline{f}(i,c)$ is conveniently bounded away from 1.

For clarity, assume for now that each $\underline{s}(i,c)$ and $\underline{f}(i,c)$ are available. We repeat the following procedure, for every i from n downto 1, and starting with $c = C$. With probability $\underline{f}(i,c)$ we include w_i in the solution, and move to entry $(i-1, c-w_i)$; with complementary probability we do not include w_i, and move to entry $(i-1,c)$. We next show how to implement this simple procedure so that it samples in $O(n \log(n/\varepsilon))$ expected time a Knapsack solution with probability different from the uniform one by a factor $(1-\varepsilon)^{\pm 1}$. The next lemma shows how to take each of the n subsequent choices.

Lemma 3. *For any $i \in \{1,\ldots,n\}$, $c \in \{0,\ldots,C\}$, in expected time $O(\log(n/\varepsilon))$ we can generate $B \in \{0,1\}$ with*

$$\left(1-\frac{\varepsilon}{n^2}\right)^i \frac{s(i-1,c-w_i)}{s(i,c)} \leq \Pr(B=0) \leq \left(1-\frac{\varepsilon}{n^2}\right)^{-i} \frac{s(i-1,c-w_i)}{s(i,c)}.$$

Proof. By Property (I_2), we get $\left(1-\varepsilon/n^2\right)^i s(i,c) \leq \underline{s}(i,c) \leq s(i,c)$. Together with (4), this implies

$$\left(1-\frac{\varepsilon}{n^2}\right)^i \frac{s(i-1,c-w_i)}{s(i,c)} \leq \underline{f}(i,c) \leq \left(1-\frac{\varepsilon}{n^2}\right)^{-i} \frac{s(i-1,c-w_i)}{s(i,c)}. \tag{11}$$

Thus, in order to generate B with the desired probability, it is enough to generate uniformly at random a number $x \in [0,1)$ and set $B=0$ iff $x < \underline{f}(i,c)$.

This can be implemented in expected time $O(\log(n/\varepsilon))$ as follows. We start generating a random sequence of bits (starting with the most significant one of x) until seeing the first bit equal to '1' (the first bit of the mantissa of x). At this point, we know the exponent of x. Since $\underline{f}(i,c)$ has a mantissa of $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits, in order to decide whether $x < \underline{f}(i,c)$, it is enough to generate other $\lceil \log(n^2/\varepsilon) \rceil$ bits for the mantissa of x. Call $\underline{x}$ the resulting floating-point number, and set $B=0$ iff $\underline{x} < \underline{f}(i,c)$.

The exponent of $\underline{x}$ can be computed by starting with the exponent equal to 0, and for every bit of x equal to 0, subtracting 1 from it. Since the expected number of bits until seeing the first bit of x equal to '1' is 2, the expected time for generating $\underline{x}$ is $O(\log(n/\varepsilon))$. □

By Lemma 3, the probability X of generating a 0/1 Knapsack solution satisfies the following relation, which by (10) gives our $(1-\varepsilon)^{\pm 1}$ approximation:

$$\left(1-\frac{\varepsilon}{n^2}\right)^{n^2} \frac{1}{s(n,C)} \leq X \leq \left(1-\frac{\varepsilon}{n^2}\right)^{-n^2} \frac{1}{s(n,C)}.$$

We show now how to implement this random generation procedure efficiently, using the lists constructed in Sec. 3. See the resulting procedure in Algorithm 2. The idea is that for every element approximating an entry $s(i,c)$, we attach one pointer to the element of $list(i-1)$ approximating $s(i-1,c)$, and one pointer to the element of $list(i-1)$ approximating $s(i-1,c-w_i)$.

We do this by extending the construction of $forw(i)$ inside Lemma 1, as follows. Assume, like in Lemma 1, that pointer $left$ is on a pair $[c,\underline{r}]$ of $list(i-1)$. Assume also that pointer $right$ reached a pair $[c_1,\underline{r}_1]$, and that we need to append the pair $[c+w_i, \underline{r}_1 \oplus \underline{r}]$ (the approximation of $s(i,c+w_i)$) at the end of $forw(i)$. We now attach to it two back-pointers: one

[5] For simplicity, in all subsequent considerations, we ignore the technical issue that we need to use one extra bit for indicating that the exponent is less than zero.

Algorithm 2: ApproximateRandomGeneration($w_1, \ldots, w_n, C$). Random generation of 0/1 Knapsack solutions.

```
compute list(1), ..., list(n) with Algorithm 1, by attaching the two back-pointers to each element;
c := C;
current := the last element of list(n);
solution := ∅;
for i := n downto 1 do
    generate B ∈ {0, 1} with Lemma 3;
    if B = 0 then
        solution := solution ∪ {w_i};
        c := c − w_i;
        update current by following its back-pointer corresponding to choosing w_i in the solution;
    else
        update current by following its back-pointer corresponding to not choosing w_i in the solution;
return solution.
```

to element $left$ (the approximation of $s(i-1, c+w_i)$) and one to element $right$ (the approximation of $s(i-1, c)$). Similarly when computing list $back(i)$. The trace back in the random generation procedure starts in the last element of $list(n)$, and follows the back-pointers corresponding to whether the current element is included or not in the solution.

The time needed to construct this collection of extended lists is the same as before, the only difference being that the floating-point numbers have mantissas of $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits, leading to a time complexity of $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil)$. The memory bound used by these lists should also take into account the space needed to store back-pointers. Each pointer must fit a computer word of W bits. Since we need back-pointers to $O(n^3 \lceil \log(n/\varepsilon) \rceil)$ entries per list, it suffices that W is $O(\lceil \log(n/\varepsilon) \rceil)$. Since we assumed operations on $O(\log(nC))$-bit numbers to take constant time, we can also assume $W = \Omega(\log(nC))$. We thus obtain that the memory needed to store all the lists is $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil W)$ bits.

We can pre-compute each $\underline{f}(i, c)$ needed in Lemma 3 using one of the two back-pointers of every element of a list, and doing the floating-point division with the Newton-Raphson division method, which reduces a division to a multiplication algorithm [10]. Thus, each division can be computed in time $O(\lceil \log(1/\varepsilon)/\log n \rceil M(\lceil \log(1/\varepsilon)/\log n \rceil)$ time, where $M(x) = \log x \log\log x$ [10] (assuming again operations on $O(\log n)$ bits to have unit cost). Generating one Knapsack solution takes expected time $O(n \log(n/\varepsilon))$, by Lemma 3. This completes the proof of Theorem 3.

5. Counting 0/1 Knapsack solutions on a DAG

Without loss of generality, we can assume that all vertices of the DAG D (with n vertices and m arcs) are reachable from s, and all vertices reach t; we also assume that the vertices are labeled in a topological order $v_1, \ldots, v_n$, such that $s = v_1$ and $t = v_n$. The dynamic programming for the 0/1 Knapsack problem can trivially be extended to a DAG. We denote by $s(i, c)$ the number of s, v_i-paths (clearly, these use a subset of the vertices $\{v_1, \ldots, v_{i-1}\}$) and of total weight at most $c \in \{0, \ldots, C\}$. If for every node v_i, its in-degree is $d(i)$, its in-neighborhood is $\{v_{i_1}, \ldots, v_{i_{d(i)}}\}$, and the weights of the arcs from each of these $d(i)$ in-neighbors are $w_{i_1}, \ldots, w_{i_{d(i)}}$, respectively, relation (6) generalizes to:

$$s(i, c) = \sum_{j=1}^{d(i)} s(i_j, c - w_{i_j}), \tag{12}$$

where we take $s(1, c) = 1$, for every $c \in \{0, \ldots, C\}$, and $s(i, c) = 0$ for every $c < 0$ and every $i \in \{1, \ldots, n\}$. The solution is obtained as $s(n, C)$. Since the number of all s, v_i-paths in the DAG is $O(2^i)$, for every $i \in \{1, \ldots, n\}$, this DP executes mC additions on n-bit numbers, and its complexity is $O(Cmn)$. Thus we can assume that $n \leq C$, and as before, that additions on $O(\log C)$-bit numbers, and thus also on $O(\log n)$-bit numbers, have unit cost.

As in our solution for the 0/1 Knapsack problem, we use dynamic programming to count, keeping at each step approximate floating-point numbers. These numbers still have $\lceil \log n \rceil$ bits for the exponent, but the length of their mantissa will be chosen based on ε, and on the number of successive floating-point additions necessary to obtain $s(n, C)$. Indeed, we will discover here below that we can organize this computation in sequences of $O(n \log(1 + \frac{m}{n}))$ repeated additions, and thus we can take the mantissa to be $1 + \lceil \log(n \log(1 + \frac{m}{n})/\varepsilon) \rceil$ bits long. Accordingly, additions and comparisons of these floating-point numbers still take the same time as before, namely $O(\lceil \log(1/\varepsilon)/\log n \rceil)$.

For clarity, we explain how we organize the computation by transforming the input DAG D into a DAG D' in which every vertex has at most two in-neighbors. For every node v_i of D, if $d(i) > 2$, we construct a complete binary tree on top of the in-neighbors $v_{i_1}, \ldots, v_{i_{d(i)}}$ of v_i, where v_i is its root; this tree has $O(d(i))$ vertices and edges, and depth $\log(d(i))$. The vertices and edges of this tree are added to D, the arcs from $v_{i_1}, \ldots, v_{i_{d(i)}}$ to v_i are removed, and all edges of the tree are directed towards v_i. Moreover, the weights of the new arcs out-going from $v_{i_1}, \ldots, v_{i_{d(i)}}$ are set to be the weights of their former arcs towards v_i; all other new arcs have weight 0. After transforming the in-neighborhood of all vertices of D, the original solutions are in one-to-one correspondence with the solutions of the transformed DAG D'.

Notice that the DAG D' has $O(m)$ vertices and $O(m)$ arcs. Moreover, since $\sum_{i=1}^{n} d(i) = m$, the length of a path in D' is at most $\max \sum_{i=1}^{n} \log d_i = \max \log \prod_{i=1}^{n} d_i$, where the maximum goes over all partitions of m into n integers $d_1, \ldots, d_n$; the maximum is obtained when all factors of the product are $\Theta(m/n)$. Thus, the length of the longest path in D' is $O(n \log(1 + \frac{m}{n}))$.

We denote by n' the number of vertices of D', and we assume that $v_1, \ldots, v_{n'}$ is a topological order on D' (so that $s = v_1$ and $t = v_{n'}$). Using the same notation as above, relation (12) simplifies to

$$s(i, c) = \begin{cases} s(i_1, c - w_{i_1}), & \text{if } d(i) = 1, \\ s(i_1, c - w_{i_1}) + s(i_2, c - w_{i_2}), & \text{if } d(i) = 2. \end{cases} \tag{13}$$

As in the case of the 0/1 Knapsack problem, for every $i \in \{1, \ldots, n'\}$, we keep a list $list(i)$ of pairs [capacity, approximate number of solutions], and use the notation $\underline{s}(i, c)$ with the same meaning. Analogously, $list(1)$ consists of the single pair $[0, 1]$, and while computing $list(i)$ from lists $list(i_1)$, or from $list(i_1)$ and $list(i_2)$, we maintain the following two invariants, where $\ell(i)$ denotes the length of the longest path from s to v_i:

(I_1) $list(i)$ is strictly increasing on both components;
(I_2) $\left(1 - \varepsilon/(n \log(1 + \frac{m}{n}))\right)^{\ell(i)} s(i, c) \leq \underline{s}(i, c) \leq s(i, c)$, for every $c \in \{0, \ldots, C\}$.

Property (I_1) implies now that the length of $list(i)$ is $O(n^2 \log(1 + \frac{m}{n})\varepsilon^{-1})$. If $d(i) = 1$, then we build $list(i)$ by scanning $list(i_1)$ and for every pair $[c_1, r_1]$, we insert the pair $[c_1 + w_{i_1}, r_1]$ in $list(i)$. It is obvious that the resulting list satisfies Properties (I_1) and (I_2).

Therefore, we consider onwards the case $d(i) = 2$. Analogously to (7), we first build a $list'(i)$ that for every capacity c_1 in $list(i_1)$, contains the pair

$$[c_1 + w_{i_1}, \underline{s}(i_1, c_1) \oplus \underline{s}(i_2, c_1 + w_{i_1} - w_{i_2})], \tag{14}$$

and for every capacity c_2 in $list(i_2)$, contains the pair

$$[c_2 + w_{i_2}, \underline{s}(i_1, c_2 + w_{i_2} - w_{i_1}) \oplus \underline{s}(i_2, c_2)]. \tag{15}$$

As in the case of Lemma 1, we can obtain $list'(i)$ in linear time from $list(i_1)$ and $list(i_2)$. By removing the dominated pairs from $list'(i)$, we obtain $list(i)$.

Lemma 4. *We can compute $list(i)$ from $list(i_1)$ and $list(i_2)$ in time $O(n^2 \log(1 + \frac{m}{n})\varepsilon^{-1} \lceil \log(1/\varepsilon) / \log n \rceil)$.*

Proof. At a generic step $i \in \{1, \ldots, n'\}$, we compute $list'(i)$ as follows. We construct two auxiliary bimonotonic lists of pairs, $list_1(i)$ and $list_2(i)$. For every capacity c_1 in $list(i_1)$, $list_1(i)$ contains the pairs $[c_1 + w_{i_1}, \underline{s}(i_1, c_1) \oplus \underline{s}(i_2, c_1 + w_{i_1} - w_{i_2})]$. Analogously, for every capacity c_2 in $list(i_2)$, $list_2(i)$ contains the pairs $[c_2 + w_{i_2}, \underline{s}(i_1, c_2 + w_{i_2} - w_{i_1}) \oplus \underline{s}(i_2, c_2)]$. List $list(i)$ is now obtained by merging in a unique list $list_1(i)$ and $list_2(i)$, and removing the dominated pairs.

In order to compute $list_1(i)$, proceed as follows. Keep two pointers, $left$ in $list(i_1)$ and $right$ in $list(i_2)$. Pointer $left$ is initially set to the first pair in $list(i_1)$, say $[c_1, r_1]$. Pointer $right$ is set to the first pair in $list(i_2)$. If $c_1 + w_{i_1} - w_{i_2} < 0$, then we append the pair $[c_1 + w_{i_1}, r_1]$ at the end of $list_1(i)$. Otherwise, pointer $right$ starts scanning $list(i_2)$ until reaching a pair $[c_2, r_2]$, such that $c_1 + w_{i_1} - w_{i_2} \geq c_2$ and either $[c_2, r_2]$ is the last pair in $list(i_2)$, or $[c_2, r_2]$ is immediately followed by a pair $[c_3, r_3]$ with the property $c_1 + w_{i_1} - w_{i_2} < c_3$. Append the pair $[c_1 + w_{i_1}, r_1 \oplus r_2]$ at the end of $list_1(i)$.

Afterwards, advance pointer $left$ to the next pair in $list(i_1)$, and repeat the above procedure, by advancing pointer $right$ to the corresponding pair, and inserting a new resulting pair in $list_1(i)$. This is repeated until pointer $left$ reaches the end of $list(i_1)$.

The computation of $list_2(i)$ is entirely analogous. Just as in the proof of Lemma 1, to obtain $list(i)$, we merge $list_1(i)$ and $list_2(i)$ on the first component, making sure that we also remove dominated pairs. Thus Property (I_1) holds for $list(i)$.

This completes the proof, since additions and comparisons on $O(\log C)$-bit numbers take unit time, floating-point additions and comparisons take $O(\lceil \log(1/\varepsilon)/\log n \rceil)$ time, and the length of $list(i_1)$ and $list(i_2)$ is $O(n^2 \log(1 + \frac{m}{n})\varepsilon^{-1})$. □

We next prove an analog of Lemma 2.

Lemma 5. *Property (I_2) holds for $list(i)$, that is, for every $i \in \{1, \ldots, n\}$ and every $c \in \{0, \ldots, C\}$, $\left(1 - \varepsilon/(n \log(1 + \frac{m}{n}))\right)^{\ell(i)} s(i, c) \leq \underline{s}(i, c) \leq s(i, c)$ holds.*

Proof. The claim is clear for $i = 1$. For an arbitrary capacity $c \in \{0, \ldots, C\}$, let $[c_0, r_0]$ in $list(i)$ be such that $\underline{s}(i, c) = r_0$. Therefore, from the definition of $\underline{s}$, we get $\underline{s}(i, c) = \underline{s}(i, c_0)$; from the fact that the pairs in $list(i)$ are of the form (14) or (15), we have

$$\underline{s}(i,c) = \underline{s}(i,c_0) = \underline{s}(i_1, c_0 - w_{i_1}) \oplus \underline{s}(i_2, c_0 - w_{i_2}). \tag{16}$$

There is no capacity c_1 in $list(i_1)$ such that $c_0 - w_{i_1} < c_1 < c - w_{i_1}$. Indeed, for assuming the contrary, $c_1 + w_{i_1}$ would be a capacity in $list'(i)$, by (14). Since we have chosen c_0 as the largest capacity in $list(i)$ smaller than c, and $c_0 < c_1 + w_{i_1} < c$ holds, this implies that $c_1 + w_{i_1}$ was pruned when passing from $list'(i)$ to $list(i)$; thus, the two pairs of $list'(i)$ having c_0 and $c_1 + w_{i_1}$ as first components have equal second components. By (16) and the bimonotonicity of $list(i_1)$, this entails that also the two pairs of $list(i_1)$ having $c_0 - w_{i_1}$ and c_1 as first components must have equal second components. This contradicts the fact that $list(i_1)$ satisfies Property (I_1). Therefore, it holds that $\underline{s}(i_1, c_0 - w_{i_1}) = \underline{s}(i_1, c - w_{i_1})$. Analogously, we get $\underline{s}(i_2, c_0 - w_{i_2}) = \underline{s}(i_2, c - w_{i_2})$.

Plugging these two relations into (16) we obtain

$$\underline{s}(i,c) = \underline{s}(i_1, c - w_{i_1}) \oplus \underline{s}(i_2, c - w_{i_2}). \tag{17}$$

From (13), the fact that Property (I_2) holds for $lists(i_1)$ and $lists(i_2)$, from (5), and since $\ell(i) = 1 + \max\{\ell(i_1), \ell(i_2)\}$, the relation above implies that

$$\left(1 - \varepsilon / \left(n \log\left(1 + \frac{m}{n}\right)\right)\right)^{\ell(i)} s(i,c) \leq \underline{s}(i,c) \leq s(i,c),$$

which shows that Property (I_2) holds also for $list(i)$. □

Notice that the transformed DAG D' has $O(m)$ vertices and the length of the longest path in D' is $O(n \log(1 + \frac{m}{n}))$. Therefore, Theorem 2 follows from Lemmas 4 and 5.

6. Conclusion

Like the technique of Štefankovič et al. [6], our results do not appear immediately generalizable to other Knapsack problems. However, our floating-point approximation layer can be applied to combinatorial decompositions of various other problems, with the required math for bounding the run-time in terms of ε embodied in this layer.

This is not only a technical layer, but also a conceptual tool that can guide and inspire the design of new algorithms. In this new scenario, the length of the mantissa becomes a resource, and minimizing its consumption leads one to reduce the number of subsequent approximation phases in processing the data flow. This view indeed supported us in gaining an extra n factor in Theorem 2. Moreover, the algorithms inspired by this framework require very little ad-hoc analysis, thanks to the reusable layer of floating-point arithmetic.

Conflict of interest statement

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

References

[1] R. Rizzi, A.I. Tomescu, Faster FPTASes for counting and random generation of Knapsack solutions, in: A. Schulz, D. Wagner (Eds.), ESA 2014 - 22nd European Symposium on Algorithms, in: Lecture Notes in Computer Science, vol. 8737, Springer-Verlag, 2014, pp. 762–773.
[2] M. Mihalák, R. Šrámek, P. Widmayer, Counting approximately-shortest paths in directed acyclic graphs, Theory Comput. Syst. 58 (1) (2016) 45–59, https://doi.org/10.1007/s00224-014-9571-7, A preliminary version appeared at WAOA 2013.
[3] M. Jerrum, L.G. Valiant, V.V. Vazirani, Random generation of combinatorial structures from a uniform distribution, Theor. Comput. Sci. 43 (1986) 169–188.
[4] P. Gopalan, A.R. Klivans, R. Meka, Polynomial-time approximation schemes for Knapsack and related counting problems using branching programs, CoRR, arXiv:1008.3187.
[5] M.E. Dyer, Approximate counting by dynamic programming, in: L.L. Larmore, M.X. Goemans (Eds.), STOC, ACM, 2003, pp. 693–699.
[6] D. Štefankovič, S. Vempala, E. Vigoda, A deterministic polynomial-time approximation scheme for counting Knapsack solutions, SIAM J. Comput. 41 (2) (2012) 356–366, https://doi.org/10.1137/11083976X.
[7] M.E. Dyer, A.M. Frieze, R. Kannan, A. Kapoor, L. Perkovic, U.V. Vazirani, A mildly exponential time algorithm for approximating the number of solutions to a multidimensional Knapsack problem, Comb. Probab. Comput. 2 (1993) 271–284.
[8] B. Morris, A. Sinclair, Random walks on truncated cubes and sampling 0-1 Knapsack solutions, SIAM J. Comput. 34 (1) (2004) 195–226.
[9] P. Gopalan, A. Klivans, R. Meka, D. Stefankovic, S. Vempala, E. Vigoda, An FPTAS for #Knapsack and related counting problems, in: R. Ostrovsky (Ed.), FOCS, IEEE, 2011, pp. 817–826.
[10] A. Schönhage, V. Strassen, Schnelle multiplikation großer zahlen, Computing 7 (3–4) (1971) 281–292, https://doi.org/10.1007/BF02242355.
[11] A. Denise, P. Zimmermann, Uniform random generation of decomposable structures using floating-point arithmetic, Theor. Comput. Sci. 218 (2) (1999) 233–248.